Formalizing Propagation of Priorities in Reo, Using Eight Colors

Formalizing Propagation of Priorities in Reo, using Eight Colors

Sung-Shik Jongmans[1,2]

[1] Department of Computer Science, Open University, the Netherlands
[2] Department of Computing, Imperial College London, United Kingdom
ssj@ou.nl

Abstract. Reo is a language for programming of coordination protocols among concurrent processes. Central to Reo are connectors: programmable synchronization/communication mediums used by processes to exchange data. Every connector runs at a clock; at every tick, it enacts an enabled synchronization/communication among processes.
Connectors may prioritize certain synchronizations/communications over others. "Passive" connectors use their priorities only at clock ticks, to decide which enabled synchronization/communication to enact. "Active" connectors, in contrast, use their priorities also between clock ticks, to influence which synchronizations/communications become enabled; they are said to "propagate their priorities".
This paper addresses the problem of formalizing propagation of priorities in Reo. Specifically, this paper presents a new instantiation of the connector coloring framework, using eight colors. The resulting formalization of propagation of priorities is evaluated by proving several desirable behavioral equalities.

Foreword

This paper addresses, perhaps, the oldest open problem in the Reo community.

The problem came to my attention for the first time in May 2011, six months into my PhD project. Perhaps—nay, surely!—I should have walked away from it; oh, the time *that* would have saved me... But, the problem was too tempting to resist. Farhad, Kasper, and I worked on solutions intermittently over the past years. Many times, I thought we had solved it; equally many times, we had not.

I promised Farhad more than once to end our suffering (my choice of words), by formalizing propagation of priorities in the connector coloring framework, using $k > 3$ colors. I never quite succeeded. This seems the perfect occasion to finally, half a decade down the road, fulfill that promise. Well, to some extent.

1 Introduction

Context. Reo is a language for programming of coordination protocols among concurrent processes. Central to Reo are *connectors*: programmable synchronization/communication mediums used by processes to exchange data, by invoking `write` and `take` operations. Every connector runs at a clock; at every tick, it enacts an enabled synchronization/communication among processes, based on the operations those processes have performed.

To send data, a process can invoke a `write` operation on the interface of a connector; to receive, it can invoke a `take` operation. Both `write`s and `take`s are *blocking*: after a process has invoked `write` or `take`, it immediately *suspends*, its operation becomes *pending*, and it *resumes* only after its operation has been *resolved* by the connector. To resolve a pending `write`, a connector performs a reciprocal `take`; to resolve a pending `take`, it performs a reciprocal `write`.

As connectors fully control resolution of pending operations, only connectors decide *when* (synchronization) and *whereto/wherefrom* (communication) data *flow*. In this way, connectors coordinate the synchronization/communication among processes.

Problem. Connectors may *prioritize* certain synchronizations/communications over others. "Passive" connectors use their priorities only *at* clock ticks, to decide which enabled synchronization/communication to enact. "Active" connectors, in contrast, use their priorities also *between* clock ticks, to influence which synchronizations/communications become enabled; they are said to "propagate their priorities".

Imagine, for instance, a connector C among processes P_1, P_2, and P_3. Imagine, moreover, that at every clock tick, C can enact either a data-flow from P_1 to P_3 with high priority (enabled only if P_1 and P_3 invoked `write` and `take`), or a data-flow from P_2 to P_3 with low priority (enabled only if P_2 and P_3 invoked `write` and `take`). If C is passive, it quietly awaits the next clock tick, checks which operations are pending to determine which data-flows are enabled (if any), chooses and enacts the one with the highest priority, and quietly awaits the next clock tick. If C is active, in contrast, it requests P_1 to invoke `write` (and P_3 to invoke `take`) *before* the next clock tick, thereby enabling C to choose and enact the high priority data-flow from P_1 to P_3 *at* the next clock tick.

Contribution. Existing formalizations of Reo do not support modeling of connectors that propagate priorities. This paper presents such a formalization.

Section 2 establishes terminology and definitions. The section is terse; more gentle introductions to Reo [Arb04,Arb11] and the connector coloring framework [CCA07,Cos10] appear elsewhere. Section 3 details the problem of formalizing propagation of priorities. Section 4 presents a solution in the connector coloring framework, using eight colors. Section 5 contains an evaluation of this solution, in terms of behavioral equalities. Section 6 concludes this paper with a discussion. Appendix A contains definitions. Proofs appear in a technical report [Jon18].

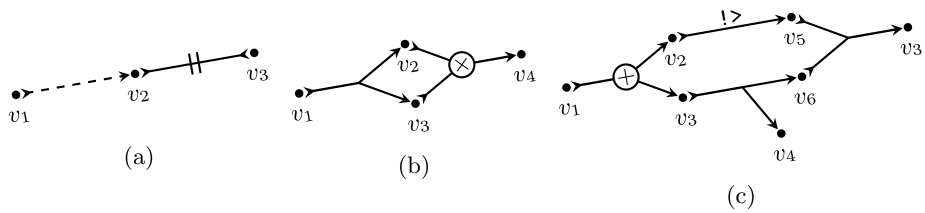


Fig. 1: Examples of connector syntax

2 Preliminaries

Syntax. Structurally, a connector in Reo is a (directed hyper) *graph* of *vertices* and (nonempty, directed hyper) *edges*.[3] Every edge is labeled with a *type*, shortly used to define the semantics of a connector. Figure 1 shows examples.

A vertex of a connector is *external* if it is the source of exactly one edge, or the target of exactly one edge; otherwise, it is *internal*. Processes perform `write` and `take` operations on external vertices, which thus consistute the interface.

A connector is *primitive*, if it has exactly one edge; otherwise, it is *compound*. Figure 2, first column, shows the name and syntax of common primitives.

A connector is *well-formed*, if (i) it has at least one edge, and (ii) if each of its vertices is the *source* of at most one edge, the *target* of at most one edge, and the source or target of at least one edge.

The structural composition of two connectors, denoted by operator $\bowtie$, is the graph consisting of the union of the sets of vertices, and the union of the sets of edges; it is a partial operation, to preserve well-formedness. Moreover, structural composition is associative and commutative.

A vertex is *shared* between two connectors, if it is an external vertex of both.

Informal semantics. Behaviorally (informal), a connector in Reo is a set of data-flows between vertices, along edges, endowed with a partial order of priorities.[4]

A vertex is *active* in a data-flow, if data passes through it; otherwise, it is *passive*. Every vertex participates either actively or passively in each of its connector's data-flows. *Idling* is the degenerate data-flow in which every vertex participates passively. A data-flow of a connector is *enabled*, if every external vertex that actively participates in the data-flow has a pending `write` or `take`; idling is always enabled, vacuously.

A connector runs on a clock; at every tick, it enacts one of its enabled data-flows. If multiple data-flows are enabled, it nondeterministically selects an order-theoretically maximal one among them. Figure 2, second column, shows the informal semantics of common primitives; "prioritizes (n) over (m)" means "(n) is greater than (m)".

---

[3] Binary edges are usually called *channels*; maximal sets of adjacent ternary edges are usually called *nodes* [Arb04,Arb11].

[4] For simplicity, and because it is a concern orthogonal to formalizing priorities, I consider only stateless connectors in this paper.

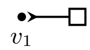
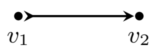
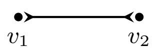
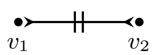
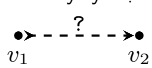
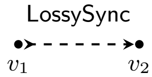
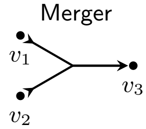
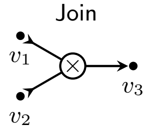
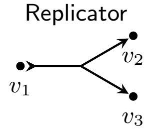
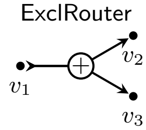
Name & Syntax	*Informal semantics*
Drain 	1. It takes data through v_1, and loses it. 2. Or, it idles. It prioritizes (1) over (2).
Sync 	1. It takes data through v_1, and writes it through v_2. 2. Or, it idles. It prioritizes (1) over (2).
SyncDrain 	1. It takes data through v_1 and v_2, and loses them. 2. Or, it idles. It prioritizes (1) over (2).
ExclDrain 	1. It takes data through v_1, and loses it. 2. Or, it takes data through v_2, and loses it. 3. Or, it idles. It prioritizes (1) over (3), and (2) over (3).
$\mathsf{LossySync}_?$ 	1. It takes data through v_1, and writes it through v_2. 2. Or, it takes data through v_1, and loses it. 3. Or, it idles. It prioritizes (1) over (3), and (2) over (3).
LossySync 	1. It takes data through v_1, and writes it through v_2. 2. Or, it takes data through v_1, and loses it. 3. Or, it idles. It prioritizes (1) over (3), and (2) over (3), and (1) over (2).
Merger 	1. It takes data through v_1, and writes it through v_3. 2. Or, it takes data through v_2, and writes it through v_3. 3. Or, it idles. It prioritizes (1) over (3), and (2) over (3).
Join 	1. It takes data through v_1 and v_2, and writes the set containing them through v_3. 2. Or, it idles. It prioritizes (1) over (2).
Replicator 	1. It takes data through v_1, and writes it through v_2 and v_3. 2. Or, it idles. It prioritizes (1) over (2).
ExclRouter 	1. It takes data through v_1, and writes it through v_2. 2. Or, it takes data through v_1, and writes it through v_3. 3. Or, it idles. It prioritizes (1) over (3), and (2) over (3).

Fig. 2: Name, syntax, and informal semantics of common primitives

$(1,1)$ It takes data through v_1, writes/takes it through v_2, and loses it.
$(2,2)$ Or, it takes data through v_1 and v_3, and loses it.
$(2,3)$ Or, it takes data through v_1 and loses it.
$(3,2)$ Or, it takes data through v_3 and loses it.
$(3,3)$ Or, it idles.
It prioritizes $(1,1)$ over $(3,3)$, and $(1,1)$ over $(2,3)$, and $(2,2)$ over $(3,3)$, and $(2,3)$ over $(3,3)$, and $(3,2)$ over $(3,3)$.

Fig. 3: Informal semantics of Fig. 1a.

A data-flow through one connector is *consistent* with a data-flow through another connector, if each of their shared vertices is either active or passive in *both* data-flows. This ensures data can flow between connectors, through their shared vertices. The behavioral composition of two connectors is the set consisting of the pairs of consistent data-flows, endowed with their product order. Every global data-flow through a compound connector, thus, is the concatenation of local data-flows.

For instance, the connector in Fig. 1a is composed of LossySync and ExclDrain in Fig. 2. As these connectors both have three local data-flows, the compound has at most nine global data-flows. Figure 3 shows which of those data-flows are consistent; (n, m) means "the pair consisting of (n) of LossySync and (m) of ExclDrain". As the compound prioritizes $(1,1)$ over $(2,3)$, and because $(1,1)$ and $(2,3)$ are *always* enabled together, it *never* enacts $(2,3)$.

Formal semantics. Behaviorally (formal), in the connector coloring framework, a connector is a set of total functions, called *colorings*, from vertices to natural numbers, called *colors* [CCA07,Cos10,JKA11,CP12]. Every coloring models a data-flow; every color models the activeness/passiveness of a vertex in a data-flow. Depending on the number of colors the framework is instantiated with, different levels of activeness/passiveness can be distinguished, to lesser or greater expressiveness. In particular, colors can be used to model priorities, as an alternative to endowing sets of colorings with partial orders (exemplified shortly).

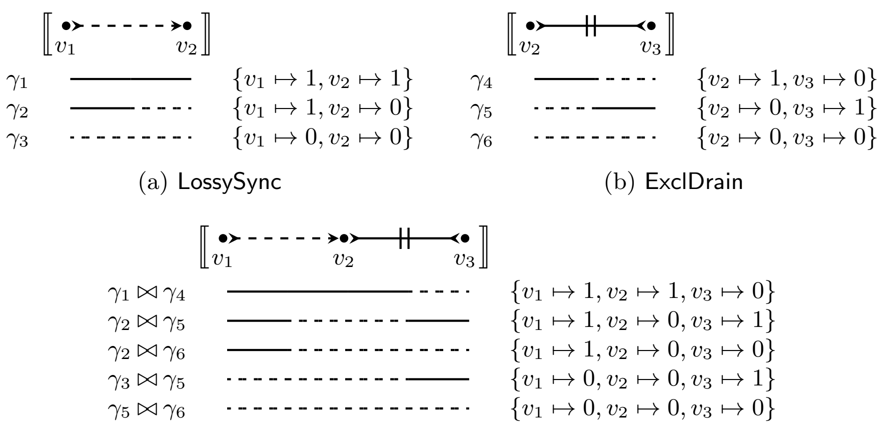
Two colorings are consistent if they map the vertices in the intersection of their domains to the same colors. The behavioral composition of two connectors, denoted by operator $\bowtie$, is the set consisting of the unions of their consistent colorings. As such, behavioral composition in the connector coloring framework straightforwardly models concatenation of consistent data-flows.[5] Behavioral composition is associative and commutative.

The structure and behavior of a connector are related through a *denotation function* $[\![\cdot]\!]$: it consumes as input a connector structure (graph) and produces as output a connector behavior (set of colorings), by decomposing the connector into primitives, looking up the local behavior of every primitive in a predefined type-indexed table, and composing the local behaviors into a global one.

[5] The composition operator can be extended with the *flip-rule* [CCA07,Cos10], to reduce sets of colorings. I do not pursue this in this paper.

#		*Meaning*
0		Passive
1		Active
2		Passive, for no `write`
3		Passive, for no `take`
4		Active; metadata-flow downstream (to propagate priorities)
5		Active; metadata-flow upstream (to propagate priorities)
6		Active; metadata-flows downstream + upstream (to propagate priorities)
7		Passive, for no `write`, for conflicting propagated priorities upstream
8		Passive, for no `take`, for conflicting propagated priorities downstream

Fig. 4: Colors

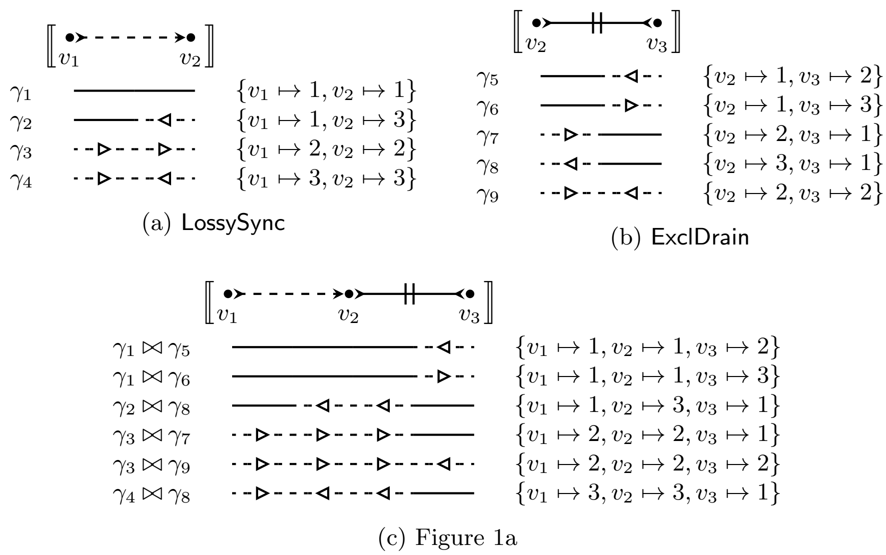


(c) Figure 1a

Fig. 5: Examples of two-color semantics

To exemplify the connector coloring framework, Fig. 4 shows nine colors. Colors $0, 1, 2, 3$ already exist in the literature; colors $4, 5, 6, 7, 8$ are new. The following lists summarizes three existing instantiations of the framework:

- With two colors [CCA07,Cos10], $\{0, 1\}$, one can model data-flows, but not priorities. Figure 5 shows examples. As the figure shows, colorings can be represented both textually and graphically (using the notation in Fig. 4).
  Figure 5a shows the behavior of LossySync. Coloring γ_1 models a data-flow from v_1 to v_2 (both vertices are active); coloring γ_2 models the loss of data taken through v_1 (only v_1 is active); coloring γ_3 models idling. Figure 5b and 5c can be explained similarly. The colorings in Fig. 5 model exactly, one-to-one, the data-flows in Figs. 2 and 3. However, priorities are not modeled.
- With three colors [CCA07,Cos10], $\{1, 2, 3\}$, one can model both data-flows and priorities. Specifically, color 0 is refined into colors $2, 3$, to model not only *that* a vertex is passive, but also *why*. Figure 6 shows examples. I write



(a) LossySync

(b) ExclDrain

(c) Figure 1a

Fig. 6: Examples of three-color semantics

"the environment can `write`/`take` through v" to mean that either a `write`/`take` is pending on v (if the environment at v is a process) or a data-flow can be concatenated at v (if the environment at v is another connector).

The expressive power of the three-color semantics is best exemplified with LossySync, as follows. Coloring γ_2 in Fig. 5a and coloring γ_2 in Fig. 6a both model the loss of data taken through v_1. However, γ_2 in Fig. 6a additionally models that this data-flow can be chosen/enacted only if no `take` can be resolved at v_2. As v_2 is a target vertex of LossySync (i.e., LossySync can only `write` through v_2), this happens only if the environment cannot `take` through v_2. Thus, if the environment can `write` through v_1, but not `take` through v_2, LossySync can lose (γ_2). But, if the environment can both `write` and `take`, LossySync must choose to not-lose (γ_1) instead of to lose (γ_2), just as its informal semantics demands (Fig 2).

The three-color semantics of $\mathsf{LossySync}_?$ is the same as the three-color semantics of LossySync, plus coloring $\gamma_2' = \{v_1 \mapsto 1, v_2 \mapsto 2\}$. This extra coloring models the loss of data taken through v_1, just as γ_2 in Fig. 6a. However, γ_2' additionally models that this data-flow can be chosen/enacted only if no `write` can be resolved at v_2. As v_2 is a target vertex of $\mathsf{LossySync}_?$ (i.e., the environment can only `take` through v_2), this happens only if $\mathsf{LossySync}_?$ cannot `write` through v_2. This is a condition that $\mathsf{LossySync}_?$ *always* can satisfy (independent of the environment). Thus, if the environment can both `write` and `take`, $\mathsf{LossySync}_?$ nondeterministically chooses between not-losing (γ_1) and losing (γ_2'); in the former case, it `write`s through v_2, while in the latter case, it does not. Thus, γ_2' is the three-color equivalent of γ_2 in Fig. 5a.

Name & Syntax	*Informal semantics*
$\mathsf{Sync}_{!>}$ $v_1 \xrightarrow{\texttt{!>}} v_2$	Same data-flows and priorities as Sync (Fig. 2). It always propagates others' priorities downstream and upstream; it always propagates its own priority downstream.
$\mathsf{Sync}_{<!}$ $v_1 \xrightarrow{\texttt{<!}} v_2$	Same data-flows and priorities as Sync (Fig. 2). It always propagates others' priorities downstream and upstream; it always propagates its own priority upstream.
$\mathsf{Sync}_{)}$ $v_1 \xrightarrow{\texttt{)}} v_2$	Same data-flows and priorities as Sync (Fig. 2). It always propagates others' priorities upstream; it never propagates priorities downstream
$\mathsf{Sync}_{(}$ $v_1 \xrightarrow{\texttt{(}} v_2$	Same data-flows and priorities as Sync (Fig. 2). It always propagates others' priorities downstream; it never propagates priorities upstream.

Fig. 7: Name, syntax, and informal semantics of priority primitives

$\mathsf{LossySync}$ and $\mathsf{LossySync}_{?}$ illustrate that by carefully modeling *why* vertices are passive, using colors $2, 3$, priorities may emerge. Graphically, the triangle markings always point away from the *root cause* for passiveness. For instance, in coloring $\gamma_3 \bowtie \gamma_7$, vertex v_2 is passive, because there is no `write` on v_2 (cause), because the environment cannot `write` through v_1 (root cause).

- With four colors [CP12], $\{0, 1, 2, 3\}$, one can model data-flows, priorities, and *partiality*. The latter is useful to allow parts of a connector to *skip* clock ticks; this is subtly different from idling, and particularly useful in distributed connector implementations. The details do not matter in this paper.

3 Problem

Informally, propagation of priorities entails the following:

> If a connector propagates the priority of a "superior" data-flow over an "inferior" one into the environment, it enacts the inferior data-flow only if: (i) another connector simultaneously propagates a priority into the environment, and (ii) the environment can facilitate only one of the two priorities—they are *conflicting*—and (iii) the environment chooses the other one. In all other cases, facilitated by the environment, the connector enacts the superior data-flow.

A connector can propagate priorities *downstream* (i.e., in the direction of data-flow), *upstream*, or in both directions.

The problem of formalizing propagation of priorities is perhaps best studied in terms of concrete connectors. To this end, the presentation of Reo so far is extended, as follows. First, Figure 7 shows four new foundational primitives that *start* ($\mathsf{Sync}_{!>}$ and $\mathsf{Sync}_{<!}$) and *end* ($\mathsf{Sync}_{)}$ and $\mathsf{Sync}_{(}$) propagation of priorities. Second, the informal semantics of every primitive in Figure 2 is extended with:

> "It always propagates others' priorities downstream and upstream, but never its own."

4 Solution

Idea. The idea is to decompose the abstract concept of propagation of priorities into two more concrete auxiliary *metadata-flows*: one from a connector to the environment and one from the environment to the connector. Through the former, called *propagation metadata-flow*, a connector informs its environment on which shared vertices the environment *must* perform reciprocal `write`s and `take`s to facilitate the propagated priority of the connector; through the latter, called *conflict metadata-flow*, the environment informs the connector on which shared vertices it *cannot* perform reciprocal `write`s and `take`s, due to conflicting propagated priorities. The direction of metadata-flows is completely independent of the direction of data-flows: metadata can flow both upstream and downstream, whereas data can flow only downstream.

Now, the plan is to model metadata-flows using colors. The problem is that metadata-flows conceptually *precede* normal data-flows (i.e., they happen between clock ticks), which cannot be directly modeled in the connector coloring framework (i.e., the framework only models what happens at clock ticks). The solution is to conflate metadata-flows and normal data-flows.

To model propagation metadata-flows from a connector to the environment, I introduce three new activeness colors: $4, 5, 6$ (Fig. 4). In a coloring, entry $v \mapsto 4$ ($v \mapsto 5$) models that vertex v *is* active in the current data-flow, and *was* active in the preceding propagation metadata-flow downstream (upstream). To model metadata-flows from the environment to the connector, I introduce two new passiveness colors: $7, 8$ (Fig. 4). In a coloring, entry $v \mapsto 7$ ($v \mapsto 8$) models that vertex v is passive in the current data flow, but *was* active in the preceding conflict metadata-flow downstream (upstream); this means the environment cannot `write` (`take`) through v, because of conflicting priorities upstream (downstream). Thus, the new instantiation of the connector coloring framework has eight colors: $\{1, 2, 3, 4, 5, 6, 7, 8\}$.

Priority primitives. Figure 8 shows the eight-color semantics of the new, priority primitives. Coloring γ_1 of $\mathsf{Sync}_{!>}$ models a data-flow from v_1 to v_2, preceded by a propagation metadata-flow downstream from v_2 (into the environment). Through this metadata-flow, $\mathsf{Sync}_{!>}$ informs the environment that it must perform a reciprocal `take` on v_2. Coloring γ_2 is similar to γ_1, except that the metadata-flow does not start at v_2, but further upstream; the metadata simply flows from v_1 to v_2. Coloring γ_3 is similar to γ_1, but beside modeling a propagation metadata-flow downstream from v_2 (into the environment), it also models a propagation metadata-flow upstream from v_2 to v_1. Coloring γ_4 combines γ_2

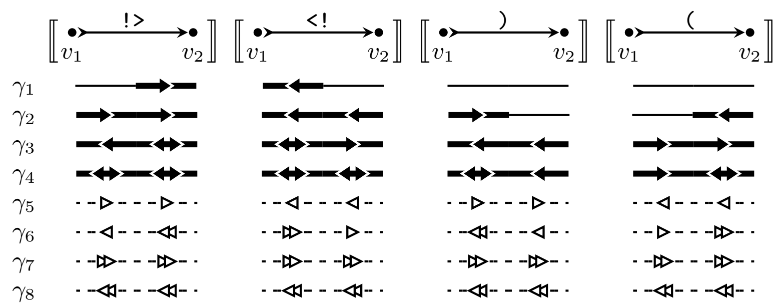


Fig. 8: Eight-color semantics of priority primitives

and γ_3. Colorings γ_5–γ_8 all model idling. Specifically, γ_5 and γ_7 permit idling if the environment cannot `write` through v_1, while γ_6 and γ_8 permit idling if the environment cannot `take` through v_2 because of conflicting propagated priorities. Note that there is no coloring that permits idling if the environment cannot `take` through v_2, *not* because of conflicting propagated priorities. The colorings of $\mathsf{Sync}_{<!}$ are symmetric.

The key colorings of $\mathsf{Sync}_{)}$ are γ_2, γ_3, and γ_6. Coloring γ_2 models a data-flow from v_1 to v_2, preceded by a propagation metadata-flow downstream to v_1, *but no further*. In this way, $\mathsf{Sync}_{)}$ *blocks* propagation of priorities downstream. Coloring γ_3 models a data-flow from v_1 to v_2, preceded by a propagation metadata-flow upstream from v_2 to v_1. This shows that the blockade works only in one direction. Coloring γ_6 models idling, *supposedly* caused by conflicting propagated priorities. However, such a conflict does not really exist: $\mathsf{Sync}_{)}$ only *pretends* it has a conflict, to enable anyone further downstream to truly ignore priorities propagated through v_1, as part of its blockade. The colorings of $\mathsf{Sync}_{(}$ are symmetric.

Common primitives. Figure 9 shows the eight-color semantics of the existing, common primitives (unary and binary); a "+M" annotation below a coloring means that the "horizontally mirrored" version of that coloring is part of the semantics as well. I highlight two salient aspects. First, the three-color semantics of every primitive [CCA07,Cos10] is strictly contained in its eight-color semantics (cf. the three-color semantics of ExclDrain and LossySync in Fig. 6). Second, coloring γ_4 of ExclDrain is a premier example of a propagation metadata-flow (from connector to environment) that induces a conflict metadata-flow (from environment to connector).

Figure 9 shows the eight-color semantics of the existing, common primitives (ternary). Again, the eight-color semantics strictly contain the three-color semantics. The interesting colorings are γ_6, γ_{16}, and γ_{12}–γ_{14} of Merger. Coloring γ_6 and γ_{16} are similar to coloring γ_4 of ExclDrain. Colorings γ_{12}–γ_{14} are notable, because they model propagation metadata-flows, but no conflict metadata-flows, in contrast to colorings γ_6 and γ_{16}. This is because propagation metadata-flows upstream have no bearing on the choices made by Merger: regardless of whether

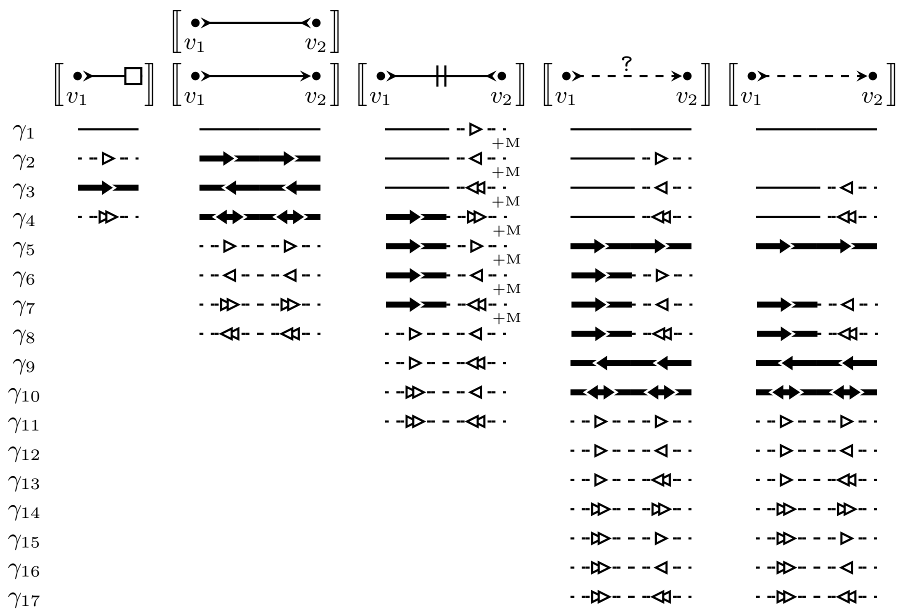


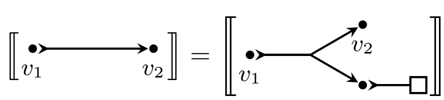
Fig. 9: Eight-color semantics of common unary and binary primitives

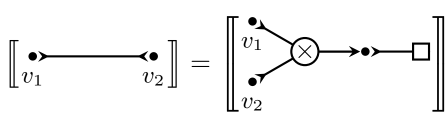
Merger chooses one of γ_{12}–γ_{14}, or one of their "vertically mirrored" versions, shared vertex v_3 is *always* active; this is all the propagated priority needs.

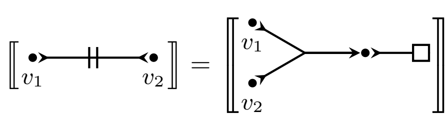
Next, to evaluate whether the eight-color semantics of the primitives compose as expected, I state and prove a number of eight-color semantics equalities.

5 Evaluation

Basic properties of common primitives. The following four propositions state that the common binary primitives in Fig. 2 (except LossySync) can be constructed out of unary and ternary primitives.[6]

Proposition 1. 

Proposition 2. 

Proposition 3. 

[6] All propositions in this paper should be interpreted modulo application of an *hide operator*, to remove internal vertices from the domains of colorings. This is straightforward to explicitly formalize.

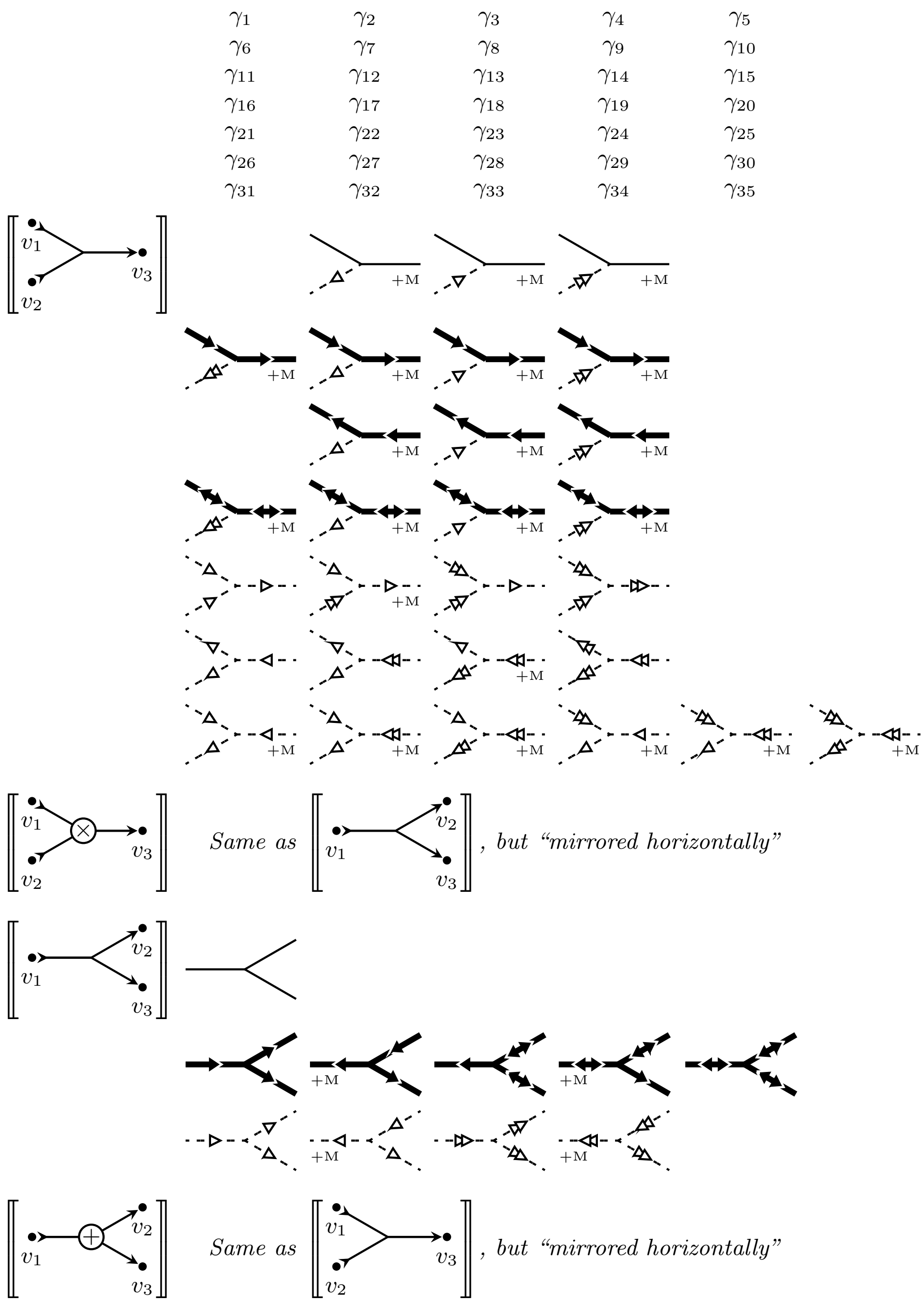
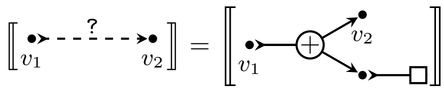
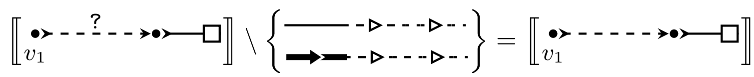


Fig. 10: Eight-color semantics of common ternary primitives

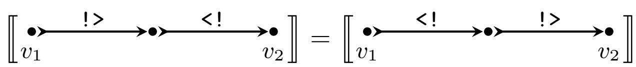
Proposition 4. 

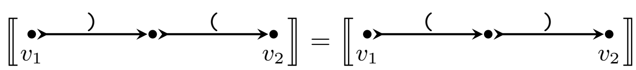
The following proposition states that LossySync$_?$ and LossySync behave differently when composed with Drain. Specifically, according to its eight-color semantics, LossySync$_?$ can (nondeterministically choose to) lose data before it reaches Drain, which LossySync cannot. This difference in semantics is intended: LossySync prioritizes not-losing over losing, whereas LossySync$_?$ does not.

Proposition 5.

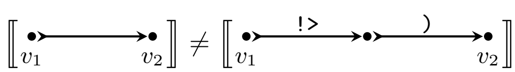


Basic properties of priority primitives. The following two propositions state that Sync$_{!>}$ and Sync$_{<!}$, and Sync$_)$ and Sync$_($, commute. Both compounds have the same data-flows and priorities as Sync in Fig. 2. But, the former compound always propagates priorities downstream and upstream, whereas the latter compound connector, in contrast, never propagates priorities downstream or upstream.

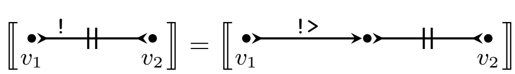
Proposition 6. 

Proposition 7. 

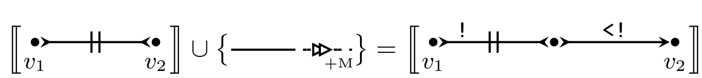
The following proposition states that Sync$_)$ is not the "inverse" of Sync$_{!>}$: starting and ending propagation of priorities is not "neutral". The reason is that Sync$_)$ ends the downstream propagation of *all* priorities; not just those of Sync$_{!>}$.

Proposition 8. 

Imagine a variant of ExclDrain that, informally, has the same data-flows and priorities as ExclDrain in Fig. 2, but additionally prioritizes (1) over (2). The following proposition states that this connector, called ExclDrain$_!$ in Fig. 11, can be constructed out of Sync$_{!>}$ and ExclDrain.

Proposition 9. 

The following proposition states that conflicting propagated priorities "cancel out": the composition of ExclDrain$_!$ and Sync$_{<!}$ is almost the same as ExclDrain. The only difference is that the compound is *saturated*: the extra coloring (cf. ExclDrain) means that the compound can always ignore propagated priorities, by pretending there is a conflict. As a result, it is actually impossible to (re)construct ExclDrain$_!$ from the compound.

Proposition 10. 

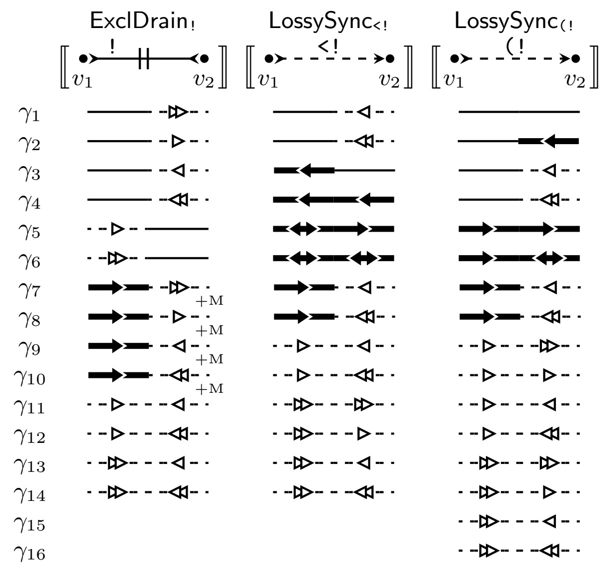
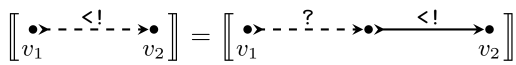


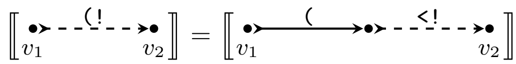
Fig. 11: Eight-color semantics of additional priority primitives

Advanced properties: context-sensitivity. Perhaps *the* litmus test for any formalization of propagation of priority is the construction of the context-sensitive LossySync out of the nondeterministic LossySync? and the priority primitives.

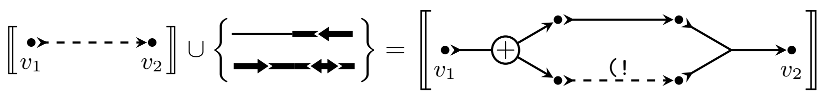
The construction proceeds in three steps. First, compose LossySync? and Sync<!. The idea is that, through propagation of its own priorities, the latter forces the former to prioritize not-losing over losing. This works, but there is an undesirable side effect: the compound connector, called LossySync<! in Fig. 11, propagates its own priorities upstream, which LossySync? does not. To solve this, second, compose Sync(and LossySync<!. The idea is that the former blocks the upstream propagation of LossySync<!'s priorities. This works, but there is again an undesirable side effect: the compound connector, called LossySync(! in Fig. 11, blocks the upstream propagation of *all* priorities (cf. Prop. 15). To solve this, finally, compose LossySync(! with ExclRouter and Merger. The idea is that the upstream propagation of others' priorities is not blocked, essentially because the propagation can proceed via a different upstream path through the graph.

The following propositions state that using the eight-color semantics, this construction *roughly* works: the only discrepancy is the presence of two colorings in the eight-color semantics of the final compound—absent in the eight-color semantics of LossySync—that model partial metadata-flows. This is an interesting phenomenon: relative to the informal semantics, the colorings are not wrong. They essentially mean that it is not really necessary to propagate priorities upstream, if a data-flow from vertex v_1 to vertex v_2 is *already* possible without such propagation. Through the construction of LossySync, this property "incidentally" emerges. I conjecture that if this property is consistently included in the eight-color semantics of all primitives, including LossySync, the resulting formalization of propagation of priority fully passes this litmus test.

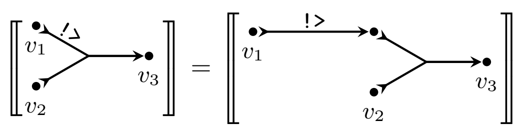
Proposition 11. 

Proposition 12. 

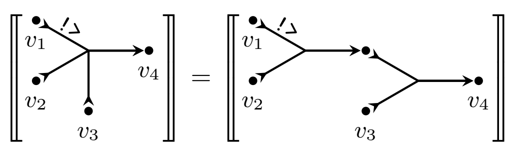
Proposition 13.



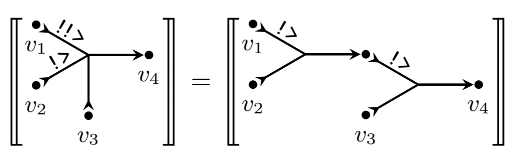
Advanced properties: ranks. Imagine a variant of Merger that, informally, has the same data-flows and priorities as Merger in Fig. 2, but additionally prioritizes (1) over (2). The following proposition states that this primitive, called $\mathsf{Merger}_{!>}$, can be composed out of $\mathsf{Sync}_{!>}$ and Merger (cf. Prop. 8).

Proposition 14. 

Imagine a variant of Merger with three sources instead of two. Informally, it has a data-flow from each of its sources to its targets, one of which it prioritizes over the other two. The following proposition states that this primitive, called $\mathsf{Merger3}_{!>}$, can be composed out of $\mathsf{Merger}_{!>}$ and Merger.

Proposition 15. 

Imagine a variant of $\mathsf{Merger3}_{!>}$ with three sources instead of two. Informally, it has a data-flow from each of its sources to its targets, one of which it prioritizes over the other two (rank #1), and one of those two (rank #2) of which it prioritizes over the other one (rank #3). The following proposition states that this primitive, called $\mathsf{Merger3}_{!!>,!>}$, can be composed out of $\mathsf{Merger}_{!>}$ and $\mathsf{Merger}_{!>}$.

Proposition 16. 

6 Discussion

I conclude this paper with some open issues and future work. Section 5 revealed already one open issue, namely the minor discrepancy between LossySync the primitive and LossySync the compound.

A second issue with the current formalization is exemplified by the connector in Fig. 1b: the eight-color semantics of this compound contains only one coloring that models idling, and moreover, this coloring has a *causality loop* (i.e., it is non-constructive, in Costa's sense [Cos10]). This problem is surprisingly difficult to solve in a proper way; the obvious solution (adding coloring $\{v_1 \mapsto 3, v_2 \mapsto 3, v_3 \mapsto 3\}$) has quite adverse side effects. Perhaps the problem can be solved by adding one or more colors.

The eight-color semantics of the connector in Fig. 1c allows for a nondeterministic choice between an "upper" data-flow (from v_1 to v_3) and a "lower" data-flow (from v_1 to v_4 and v_3), because $\mathsf{Sync}_{!>}$'s priorities are propagated only downstream, not affecting the nondeterministic choice of ExclRouter, upstream. This is a reasonable interpretation of the informal semantics. An alternative interpretation, and arguably equally reasonable, is that Merger should propagate priorities from v_5 not only to v_3 but also to v_6, reversing the direction of propagation from downstream to upstream. Under this interpretation, the nondeterministic choice of ExclRouter *is* affected by $\mathsf{Sync}_{!>}$'s priorities, and the lower data-flow should never be chosen. It would be interesting to investigate how to model this alternative interpretation in the connector coloring framework.

Finally, the eight-color semantics of primitives and compounds quickly become prohibitively large. This makes manually reasoning about these semantics quite challenging. The development of software tooling to automate the composition of sets of colorings is imperative to continue this line of research.

References

Arb04. Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

Arb11. Farhad Arbab. Puff, the magic protocol. In *Formal Modeling: Actors, Open Systems, Biological Systems*, volume 7000 of *Lecture Notes in Computer Science*, pages 169–206. Springer, 2011.

CCA07. Dave Clarke, David Costa, and Farhad Arbab. Connector colouring I: synchronisation and context dependency. *Sci. Comput. Program.*, 66(3):205–225, 2007.

Cos10. David Costa. *Formal Models for Component Connectors*. PhD thesis, Vrije Universiteit, 2010.

CP12. Dave Clarke and José Proença. Partial connector colouring. In *COORDINATION*, volume 7274 of *Lecture Notes in Computer Science*, pages 59–73. Springer, 2012.

JKA11. Sung-Shik T. Q. Jongmans, Christian Krause, and Farhad Arbab. Encoding context-sensitivity in reo into non-context-sensitive semantic models. In *COORDINATION*, volume 6721 of *Lecture Notes in Computer Science*, pages 31–48. Springer, 2011.

Jon18. Sung-Shik Jongmans. Formalizing Propagation of Priorities in Reo, using Eight Colors (Technical Report). Technical report, 2018. `www.arxiv.org`.

A Definitions

Definition 1 (Structure). $\mathbb{V}$ *is the set of all vertices.* $\mathbb{T}$ *is the set of all types. The structure of a connector is a tuple* $g = (V, E)$*, where* $V \subseteq \mathbb{V}$ *and* $E \subseteq (2^V \times \mathbb{T} \times 2^V) \setminus \{(\emptyset, t, \emptyset) \mid t \in \mathbb{T}\}$. $\mathbb{G}$ *is the set of all structures.*

Definition 2 (Structural composition). $\mathsf{S}, \mathsf{T} : 2^{(2^{\mathbb{V}} \times \mathbb{T} \times 2^{\mathbb{V}})} \to 2^{\mathbb{V}}$ *are the functions defined by the following equations:*

$$\begin{aligned} \mathsf{S}(E) &= \bigcup\{V \mid (V, t, V') \in E\} \setminus \bigcup\{V' \mid (V, t, V') \in E\} \\ \mathsf{T}(E) &= \bigcup\{V' \mid (V, t, V') \in E\} \setminus \bigcup\{V \mid (V, t, V') \in E\} \end{aligned}$$

$\bowtie : \mathbb{G} \times \mathbb{G} \to \mathbb{G}$ *is the partial operation defined by the following equation:*

$$(V_1, E_1) \bowtie (V_2, E_2) = \begin{cases} (V_1 \cup V_2, E_1 \cup E_2) & \text{if: } \mathsf{S}(E_1) \cap \mathsf{T}(E_2) = \mathsf{S}(E_2) \cap \mathsf{T}(E_1) \\ \bot & \text{otherwise} \end{cases}$$

Definition 3 (Behavior). $\mathbb{C}$ *is the set of all colors. A coloring* γ *over* V *is a function from* V *to* $\mathbb{C}$. $\mathbb{C}\textsc{ol}(V) = V \to \mathbb{C}$ *is the set of all colorings over* V*. The behavior of a connector* (V, E) *is a set* $\Gamma \subseteq \mathbb{C}\textsc{ol}(V)$ *of colorings.*

Definition 4 (Behavioral composition).
$\bowtie : (\mathbb{C}\textsc{ol}(V_1) \times \mathbb{C}\textsc{ol}(V_2) \rightharpoonup \mathbb{C}\textsc{ol}(V_1 \cup V_2)) \cup (2^{\mathbb{C}\textsc{ol}(V_1)} \times 2^{\mathbb{C}\textsc{ol}(V_2)} \rightharpoonup 2^{\mathbb{C}\textsc{ol}(V_1 \cup V_2)})$ *is the partial function defined by the following equations:*

$$\gamma_1 \bowtie \gamma_2 = \begin{cases} \gamma_1 \cup \gamma_2 & \text{if: } \gamma_1(p) = \gamma_2(p) \text{ for-all } p \in \mathit{dom}(\gamma_1) \cap \mathit{dom}(\gamma_2) \\ \bot & \text{otherwise} \end{cases}$$

$$\Gamma_1 \bowtie \Gamma_2 = \{\gamma_1 \bowtie \gamma_2 \mid \gamma_1 \in \Gamma_1 \text{ and } \gamma_2 \in \Gamma_2 \text{ and } \gamma_1 \bowtie \gamma_2 \in \mathit{dom}(\bowtie)\}$$

Definition 5 (Denotation). *With* $\mathcal{T} : \mathbb{T} \to (2^{\mathbb{V}} \times 2^{\mathbb{V}}) \to \bigcup\{2^{\mathbb{C}\textsc{ol}(V)} \mid V \subseteq \mathbb{V}\}$, $[\![\cdot]\!] : \mathbb{G} \to \bigcup\{\mathbb{C}\textsc{ol}(V) \mid V \subseteq \mathbb{V}\}$ *is the function defined by the following equation:*

$$[\![(V, E)]\!] = \bigotimes\{\mathcal{T}(t)(V, V') \mid (V, t, V') \in E\}$$

Theorem 1. $[\![g_1 \bowtie g_2]\!] = [\![g_1]\!] \bowtie [\![g_2]\!]$